



Design Patterns

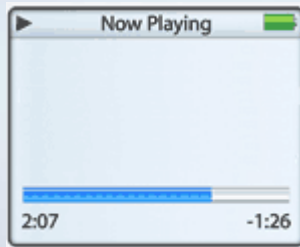
Gerwin van Dijken (gerwin.vandijken@inholland.nl)

Program term 1.4

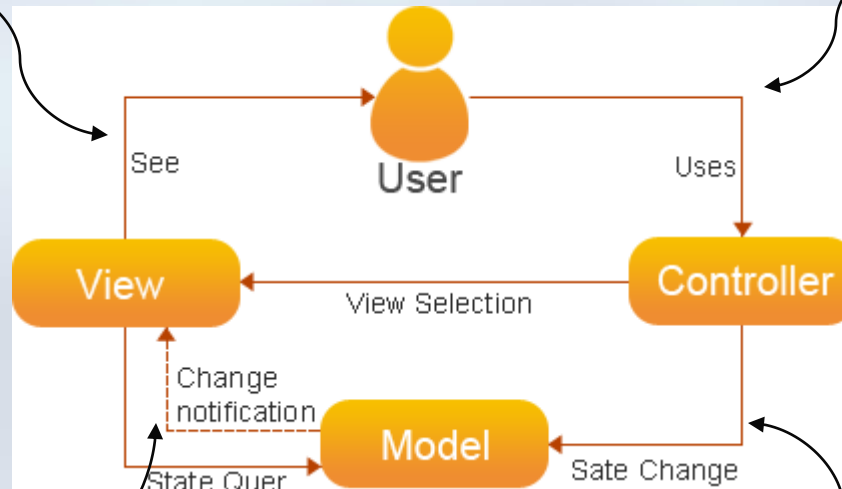
01 (wk-16)	abstract classes and interfaces
02 (wk-17)	Template Method pattern / Observer pattern
03 (wk-18)	'voorjaarsvakantie' (spring break)
04 (wk-19)	MVC pattern
05 (wk-20)	Strategy pattern / Adapter pattern
06 (wk-21)	Singleton pattern / State pattern
07 (wk-22)	Factory patterns
08 (wk-23)	repetition / practice exam
<hr/>	
09 (wk-24)	exam (<i>computer assignments</i>)
10 (wk-25)	<i>retakes (courses term 1.3)</i>
11 (wk-26)	<i>retakes (courses term 1.4)</i>

MVC – Model View Controller

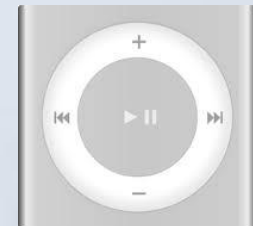
User sees the new state of the model via the View(s).



Model notifies the View that the state has changed ('a new song has started').



User uses the Controller to change the model ('start a new song').



Controller passes on the action to the Model ('a new song has to be started').

```
class MP3Player
{
    private List<Song> playList;
    private Song currentSong;
    // ...

    public void Play() { }
    public void Stop() { }
    public void Next() { }
    public void SetVolume(int level) { }
}
```

An example application

- **[model]** an MP3 player, containing a playlist, and functions like Play / Stop / Next
- **[controller]** a controller to control the MP3 player (play/stop, next, volume)
- **[views]** a display to show the current song, and a separate display to show the volume

Model: MP3 player

- We will use an interface for the model
- The controller and view know this interface

```
public interface IMP3Player
{
    void Play();
    void Stop();
    void Next();
    void SetVolume(int volumeLevel);

    Song CurrentSong { get; }
    bool IsPlaying { get; }
    int VolumeLevel { get; }

    void AddObserver(ISongObserver observer);
    void RemoveObserver(ISongObserver observer);
    void AddObserver(IVolumeObserver observer);
    void RemoveObserver(IVolumeObserver observer);
}
```

There are some read-only properties in the interface.

The MP3 player can be controlled by using these methods.

These methods are needed for adding/removing observers. We use 2 different kinds of observers.

Model: 'the real thing'

(1/3)

```
class MP3Player : IMP3Player
{
    private List<Song> playList;
    private Song currentSong;
    private bool isPlaying;
    private int volumeLevel;

    private List<ISongObserver> songObservers;
    private List<IVolumeObserver> volumeObservers;

    public Song CurrentSong { get { return currentSong; } }
    public bool IsPlaying { get { return isPlaying; } }
    public int VolumeLevel { get { return volumeLevel; } }

    // ...
}
```

Members of the MP3 player, this is the state! (current song/volume)

2 separate lists for the observers.

Properties of the MP3 player.

IMP3Player is the 'contract', class MP3Player must comply with this contract...

Model: 'the real thing'

(2/3)

We create the 2 observer lists in the constructor.

When the current song changes (play, stop of next), all SongObservers are notified.

When the current volume changes, all VolumeObservers are notified.

```
public MP3Player() {  
    // ...  
  
    // make observer lists  
    songObservers = new List<ISongObserver>();  
    volumeObservers = new List<IVolumeObserver>();  
}  
  
public void Play() {  
    // ...  
    NotifySongObservers();  
}  
  
public void Stop() {  
    // ...  
    NotifySongObservers();  
}  
  
public void Next()  
{  
    // ...  
    NotifySongObservers();  
}  
  
public void SetVolume(int volumeLevel)  
{  
    // ...  
    NotifyVolumeObservers();  
}
```

Model: 'the real thing'

(3/3)

Public methods for
SongObservers
(Add & Remove).

Public methods for
VolumeObservers
(Add & Remove).

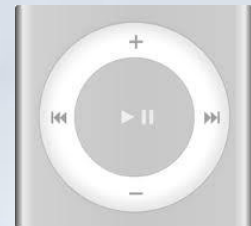
Private methods for
Notify_X_Observers.

```
// ...  
  
public void AddObserver(ISongObserver observer) {  
    songObservers.Add(observer);  
}  
  
public void RemoveObserver(ISongObserver observer) {  
    songObservers.Remove(observer);  
}  
  
public void AddObserver(IVolumeObserver observer) {  
    volumeObservers.Add(observer);  
}  
  
public void RemoveObserver(IVolumeObserver observer) {  
    volumeObservers.Remove(observer);  
}  
  
private void NotifySongObservers() {  
    foreach (ISongObserver observer in songObservers)  
        observer.Update(this.currentSong);  
}  
  
private void NotifyVolumeObservers() {  
    foreach (IVolumeObserver observer in volumeObservers)  
        observer.Update(this.volumeLevel);  
}  
}
```


Controller

- We will also use an interface (contract) for the Controller
- The Model is only manipulated by the Controller

```
public interface IMP3Controller
{
    void Play();
    void Stop();
    void Next();
    void VolumeUp();
    void VolumeDown();
}
```



Controller

MP3Controller implements IMP3Controller.

The controller receives an IMP3Player via the constructor.

Some actions are passed on 1-on-1 to the model (player).

Other actions have to be changed a bit, before passing them on to the model.

```
public class MP3Controller : IMP3Controller
{
    private IMP3Player player;

    public MP3Controller(IMP3Player player) {
        this.player = player;
    }

    public void Play() {
        player.Play();
    }

    public void Stop() {
        player.Stop();
    }

    public void Next() {
        player.Next();
    }

    public void VolumeUp() {
        int volumeLevel = player.VolumeLevel;
        player.SetVolume(++volumeLevel);
    }

    public void VolumeDown() {
        int volumeLevel = player.VolumeLevel;
        player.SetVolume(--volumeLevel);
    }
}
```

View: Observer interfaces

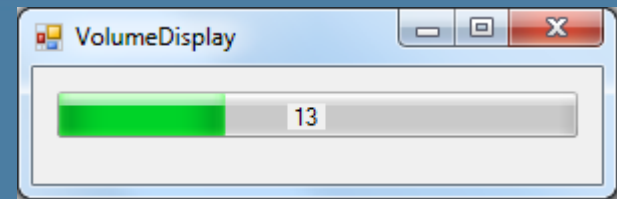
- ISongObserver: song change
- IVolumeObserver: volume change

```
public interface ISongObserver
{
    void Update(Song currentSong);
}
```

```
public interface IVolumeObserver
{
    void Update(int currentVolume);
}
```

- Could we also use one Observer interface? What would be the consequences?

View: VolumeDisplay



VolumeDisplay implements interface IVolumeObserver.

This display receives an IMP3Player via the constructor.

This display subscribes itself as a VolumeObserver...

...in order to have the Update-method called by the model, every time the volume changes.

```
public partial class VolumeDisplay : Form, IVolumeObserver
{
    IMP3Player player;

    public VolumeDisplay(IMP3Player player)
    {
        InitializeComponent();

        this.player = player;
        this.player.AddObserver(this);
    }

    private void VolumeDisplay_Load(object sender, EventArgs e)
    {
        // initialize progress bar
        progressBar1.Minimum = 0;
        progressBar1.Maximum = 40;
        progressBar1.Value = player.VolumeLevel;
        lblVolumeLevel.Text = player.VolumeLevel.ToString();
    }

    public void Update(int currentVolume)
    {
        progressBar1.Value = currentVolume;
        lblVolumeLevel.Text = currentVolume.ToString();
    }
}
```

View: MP3Display

MP3Display implements interface ISongObserver and IVolumeObserver.

```
public partial class MP3Display : Form, ISongObserver, IVolumeObserver
{
    private IMP3Player player;
    private IMP3Controller controller;

    public MP3Display(IMP3Player player, IMP3Controller controller) {
        InitializeComponent();

        this.player = player;
        this.controller = controller;

        this.player.AddObserver((ISongObserver)this);
        this.player.AddObserver((IVolumeObserver)this);
    }

    public void Update(Song currentSong) {
        // update song information
        if (currentSong == null)
            lblCurrentSong.Text = "Not playing...";
        else
            lblCurrentSong.Text =
                String.Format("{0} ({1})", currentSong.Title, currentSong.Artist);
    }

    public void Update(int currentVolume)
    {
        lblVolume.Text = currentVolume.ToString();
    }
}
```

This display receives an IMP3Player and an IMP3Controller via the constructor.

The Display subscribes itself as a SongObserver and as a VolumeObserver...

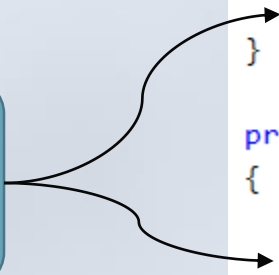
... in order to have both Update methods called by the model.

MP3Display

All user input is passed on to the controller.



Also the volume adjustment is passed on to the controller.



```
// ...

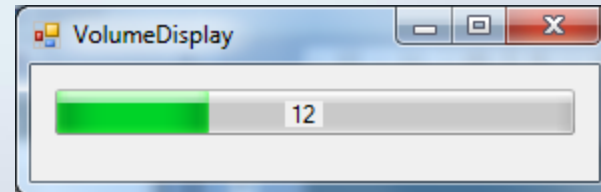
private void btnStopPlay_Click(object sender, EventArgs e)
{
    if (player.IsPlaying)
        // pass 'stop' action on to controller
        controller.Stop();
    else
        // pass 'play' action on to controller
        controller.Play();
}

private void btnNext_Click(object sender, EventArgs e)
{
    // pass 'next' action on to controller
    controller.Next();
}

private void btnVolumeUp_Click(object sender, EventArgs e)
{
    // pass 'volume up' action on to controller
    controller.VolumeUp();
}

private void btnVolumeDown_Click(object sender, EventArgs e)
{
    // pass 'volume down' action on to controller
    controller.VolumeDown();
}
```

Displays



Summary

- With the MVC pattern we separate a few things: the data/the model, the presentation of the model, the processing of user input
- The model is only manipulated by the controller
- Events (of a ControlPanel-form) are passed on (delegated) to the controller

Assignments

- BlackBoard: 'Week 3 assignments'